### Insertion Sort

<table>
<thead>
<tr>
<th>values</th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>36</td>
<td>10</td>
<td>24</td>
<td>6</td>
<td>12</td>
</tr>
</tbody>
</table>

One by one, each as yet unsorted array element is inserted into its proper place with respect to the already sorted elements. On each pass, this causes the number of already sorted elements to increase by one.

Insertion Sort works like someone who “inserts” one more card at a time into a hand of cards that are already sorted. To insert 12, we need to make room for it by moving first 36 and then 24.
Insertion Sort

Works like someone who “inserts” one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.

6 10 24
36
12

6 10 24 36
12

6 10 12 24 36
A Snapshot of the Insertion Sort Algorithm

Insertion Sort Code for Insertion Sort

```c
void InsertionSort(int values[], int numValues)
// Post: Sorts array values[0 . . . numValues-1] into ascending order by key
//
// for (int curSort=0; curSort<numValues; curSort++)
// InsertItem (values, 0, curSort);
```

```c
void InsertionSort(int values[], int numValues)
// Post: Sorts array values[0 . . . numValues-1] into ascending order by key
//
// for (int curSort=0; curSort<numValues; curSort++)
// InsertItem (values, 0, curSort);
```
Insertion Sort code (contd.)

```c
void InsertItem (int values[], int start, int end)
{   // Post: inserts values[end] at the correct position
    // into the sorted array values[start..end-1]
    bool finished = false;
    int current = end;
    bool moreToSearch = (current != start);
    while (moreToSearch && !finished)
    {   if (values[current] < values[current - 1])
            Swap(values[current], values[current - 1]);
            current--;
            moreToSearch = (current != start);
    }
    else
        finished = true;
}
```

Radix Sort

```
762
124
432
761
800
402
976
100
001
999
```

Radix Sort – Pass One

```
```

```c
Radix Sort – Pass One
```
Radix Sort – Pass Three

Radix Sort – End Pass Three

Radix Sort
Radix Sort

- Cost:
  - Each pass: n moves to form groups
  - Each pass: n moves to combine them into one group
  - Number of passes: d
  - 2*d*n moves, 0 comparison
  - However, demands substantial memory
- not a comparison sort

Sorting Algorithms and Average Case Number of Comparisons

- Radix Sort: $O(dn)$ where $d =$ max # of digits in any input number
- Simple Sorts
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
  - $O(N^2)$
- More Complex Sorts
  - Quick Sort
  - Merge Sort
  - Heap Sort
  - $O(N \log N)$

Binary Search

- Given an array $A[1..n]$ and $x$, determine whether $x \in A[1..n]$
- Compare with the middle of the array
- Recursively search depending on the result of the comparison